COP 3330: Object-Oriented Programming Summer 2007

Introduction to Classes – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

A Closer Look At Parameter Passing in Java

- Java uses the ***pass by value*** method for parameter passing.
- Whenever a method is invoked, flow of control is transferred temporarily to that method.
- The actual parameters in the invocation are used to initialize the formal parameters in the method's definition.
- For each method invocation, Java sets aside memory, known as the ***activation record*** (for that particular invocation).
- The activation record stores, among other things, the values of the formal parameters.

A Closer Look At Parameter Passing in Java (cont.)

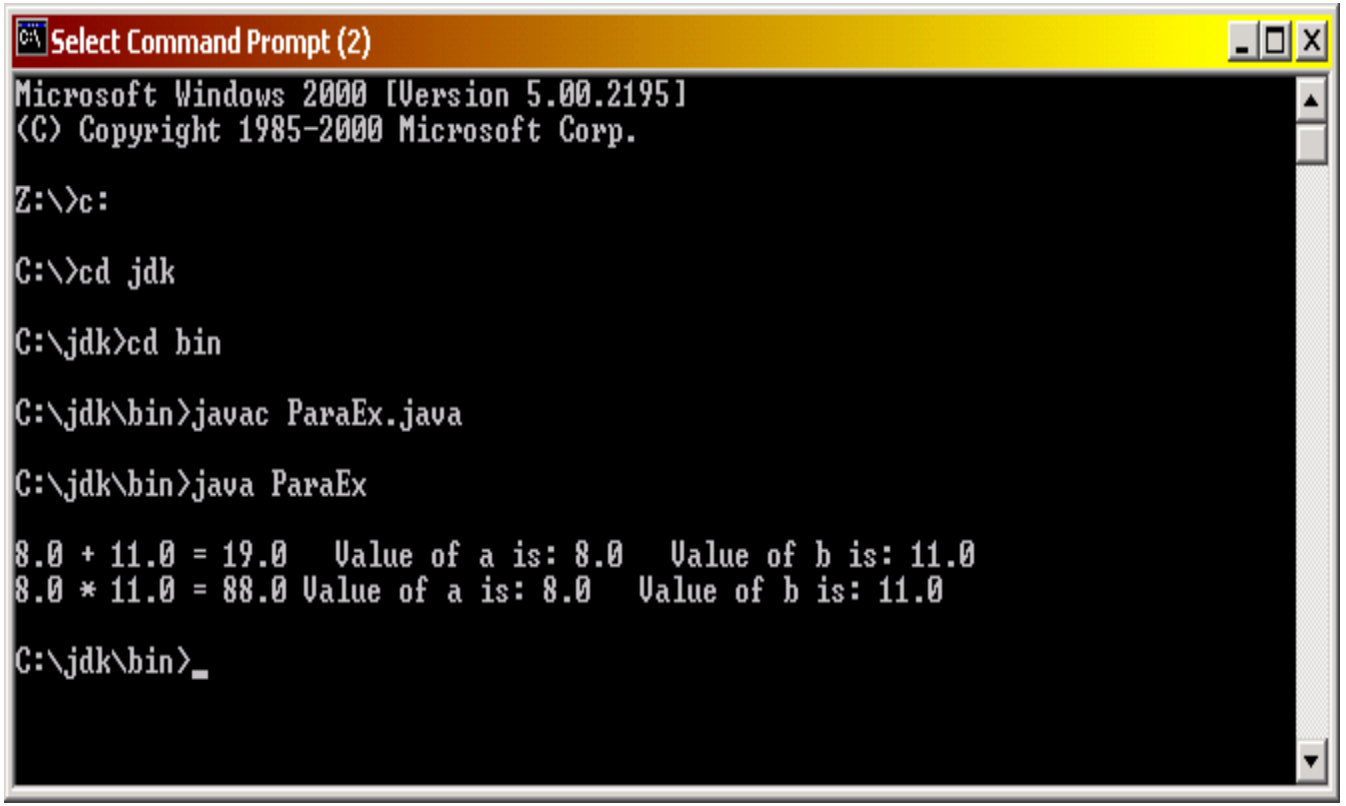
- Once the actual parameters have been used to initialize the formal parameters, the actual parameters and the formal parameters are independent of each other.

- If a method updates the value of a formal parameter, the changes **DOES NOT** affect the actual parameter. The change is limited to the activation record containing the value of the formal parameter.

Another Parameter Passing Example (1)

```
//Parameter passing example
public class ParaEx {
    //add() returns the sum of its parameters
    public static double add(double x, double y){
          double result = x + y;
          return result;
}
   //multiply() returns the product of its parameters – it is written only to
   // illustrate parameter passing.
   public static double multiply(double x, double y){
          x = x * y;
          return x;
    }
    //main() application entry point
    public static void main(String[] args){
          System.out.println();
          double a = 8;
          double b = 11;
          double sum = add(a, b);
          System.out.println(a + " + " + b + " = " + sum + "  Value of a: "
                    + a + "  Value of b: " + b);
          double product = multiply(a, b);
          System.out.println(a + " * " + b + " = " + product + "  Value of a:
                    " + a + "  Value of b: " + b);
    } }
```

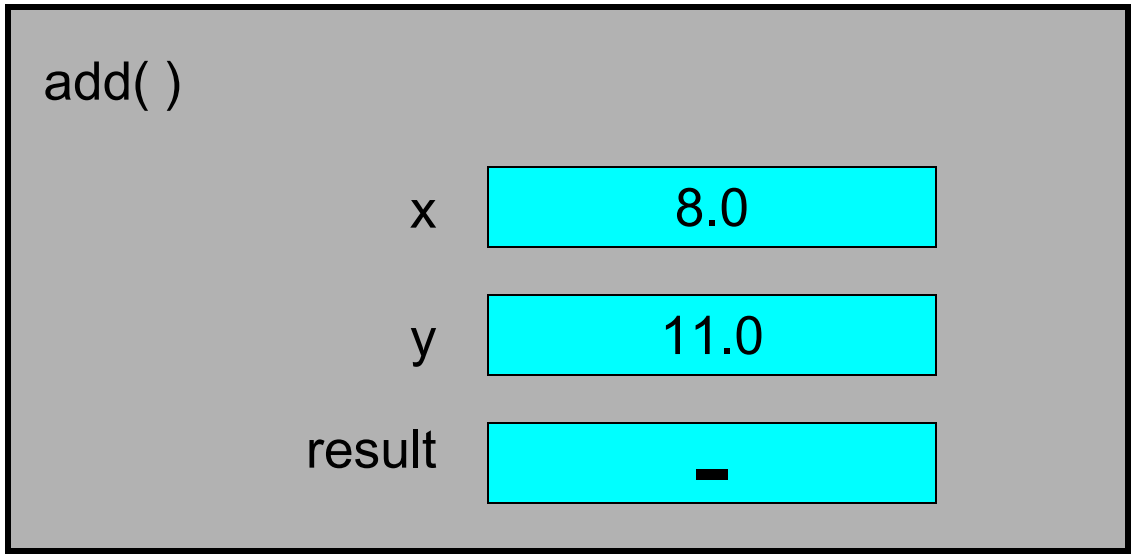

Another Parameter Passing Example – Output (1)



Another Parameter Passing Example (cont.)

Step 2:

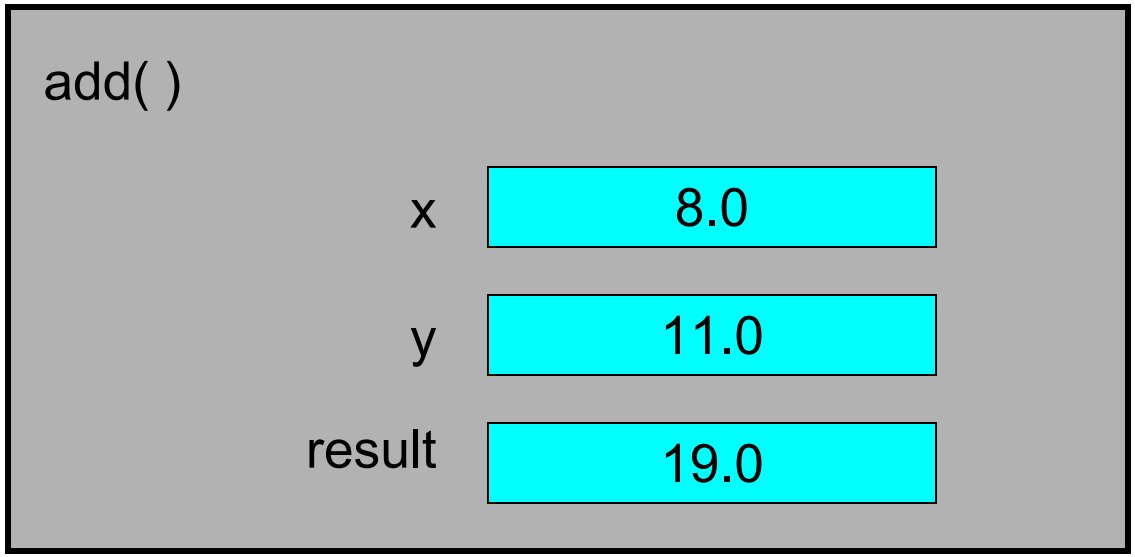
With the invocation, a new activation record is created for the method add() and the flow of control is transferred to the add() method. Creation of this activation record initializes formal parameter **x** to the value of actual parameter **a** and formal parameter **y** to the value of actual parameter **b**.



Another Parameter Passing Example (cont.)

Step 3:

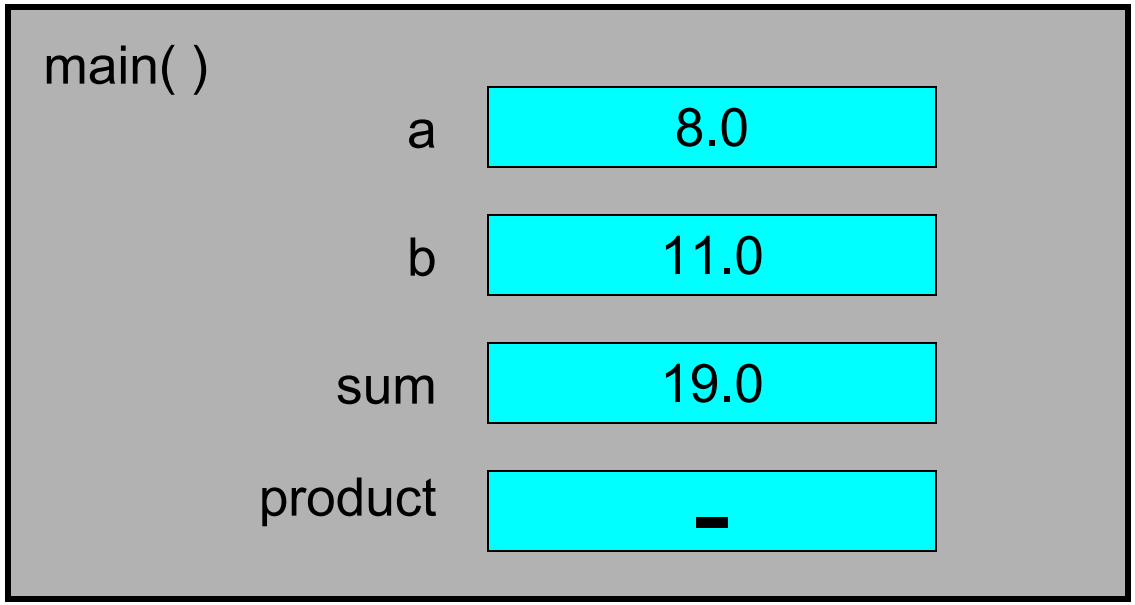
If add() updates the value of its formal parameters, the change is visible only within add() – the actual parameters are unchanged. The formal parameters **x** and **y** are used to initialize **result**. After the initialization of result the activation record for add() looks like the one shown below.



Another Parameter Passing Example (cont.)

Step 4:

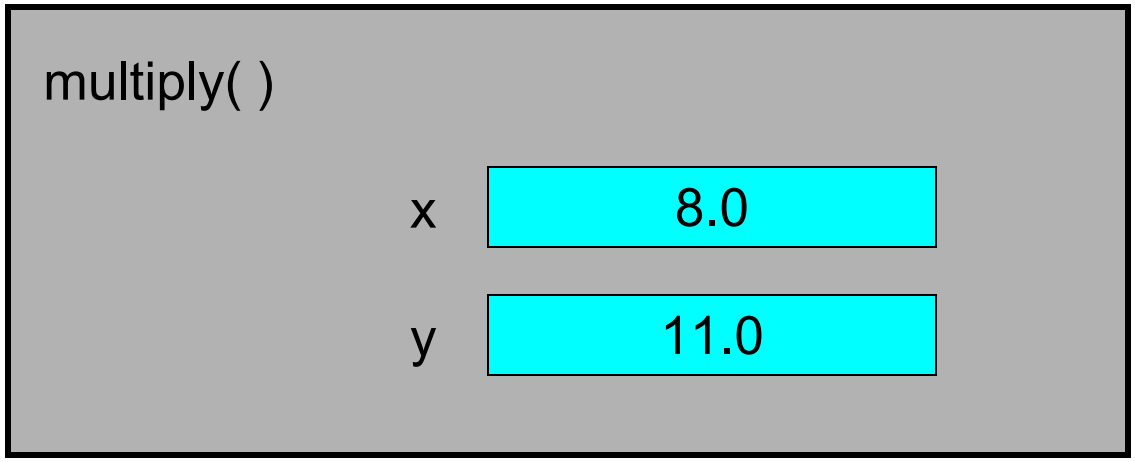
Within the add() method the return statement is evaluated. This sets the value of result to 19.0. The method's completion causes the memory associated with its activation record to be released. The flow of control is returned to the main() method, where the value returned by add() is used to initialize **`sum`**. The main() method activation record now looks like the following:



Another Parameter Passing Example (cont.)

Step 5:

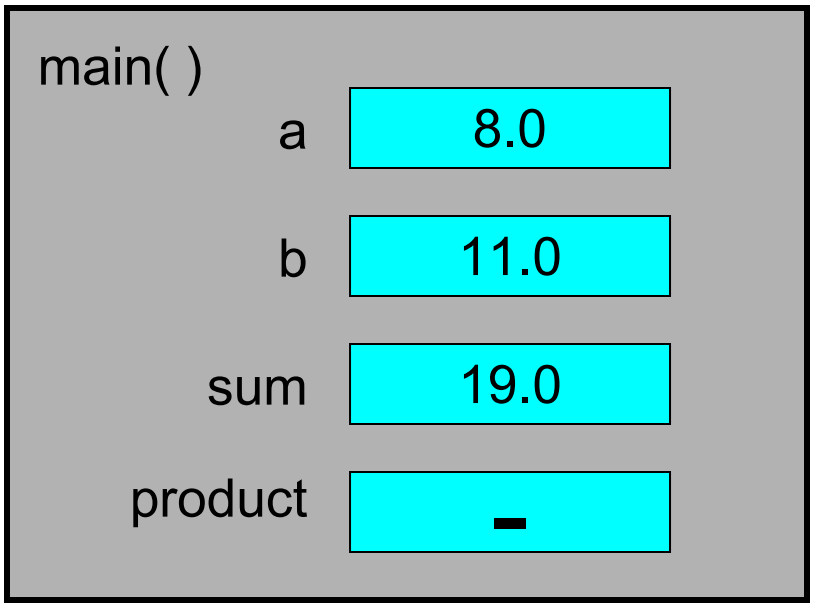
The multiply() method is invoked next which causes the flow of control to pass to this method and the creation of an activation record. The formal parameters **x** and **y** are initialized to the values of **a** and **b** respectively. The activation record for multiply() looks like the following:

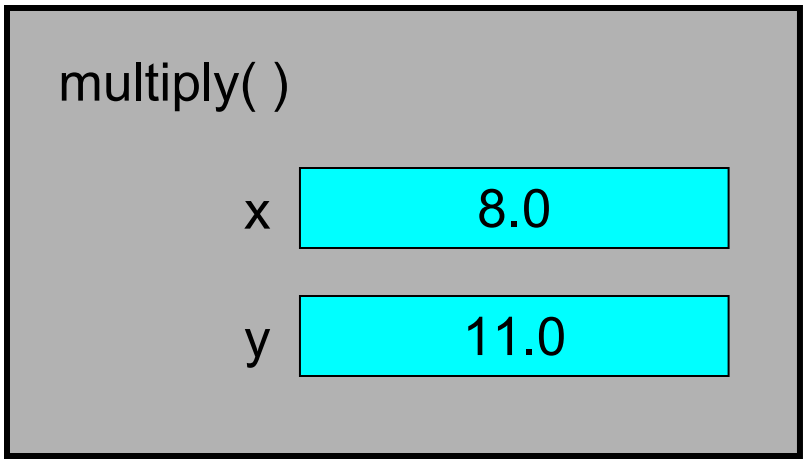


Another Parameter Passing Example (cont.)

Step 6:

After the assignment in the multiply() method, the activation records for the main() and multiply() methods are shown below.





Another Parameter Passing Example (cont.)

Step 6 - Explanation:

Because a formal parameter is essentially a variable local to its method, a change to a formal parameter does not affect the corresponding actual parameter.

The assignment of formal parameter **x** of multiply() did not affect actual parameter **a** of main().

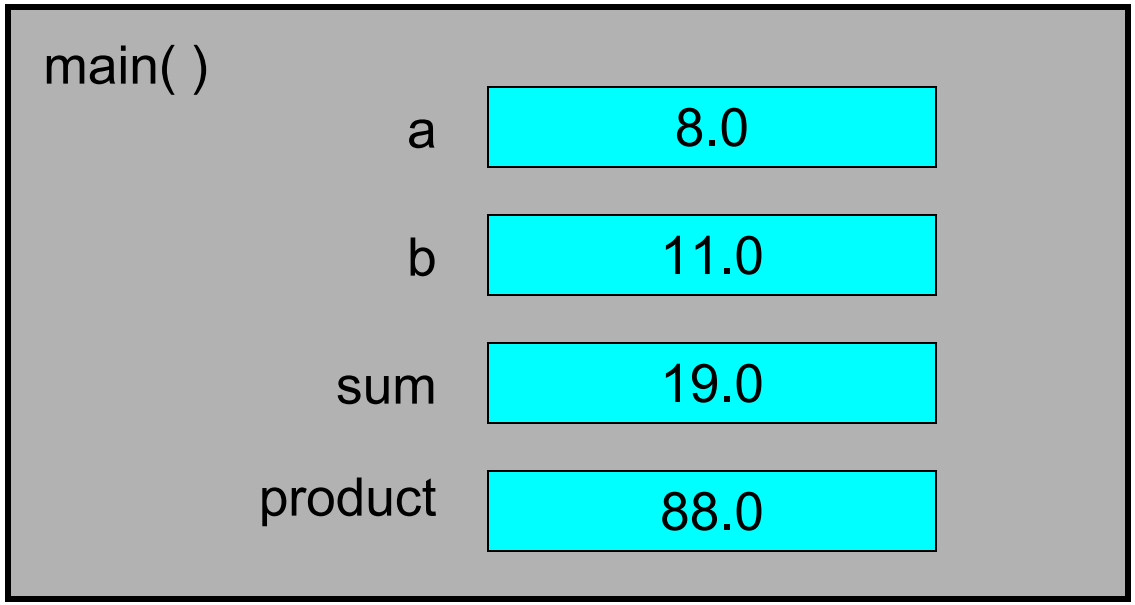
Formal parameters are created with the invocation of their method, their values are accessible only within the method, and they are destroyed when their method completes.

The only difference between the formal parameters and the other variables defined by a method is that the formal parameters are initialized using the values of the actual parameters.

Another Parameter Passing Example (cont.)

Step 7:

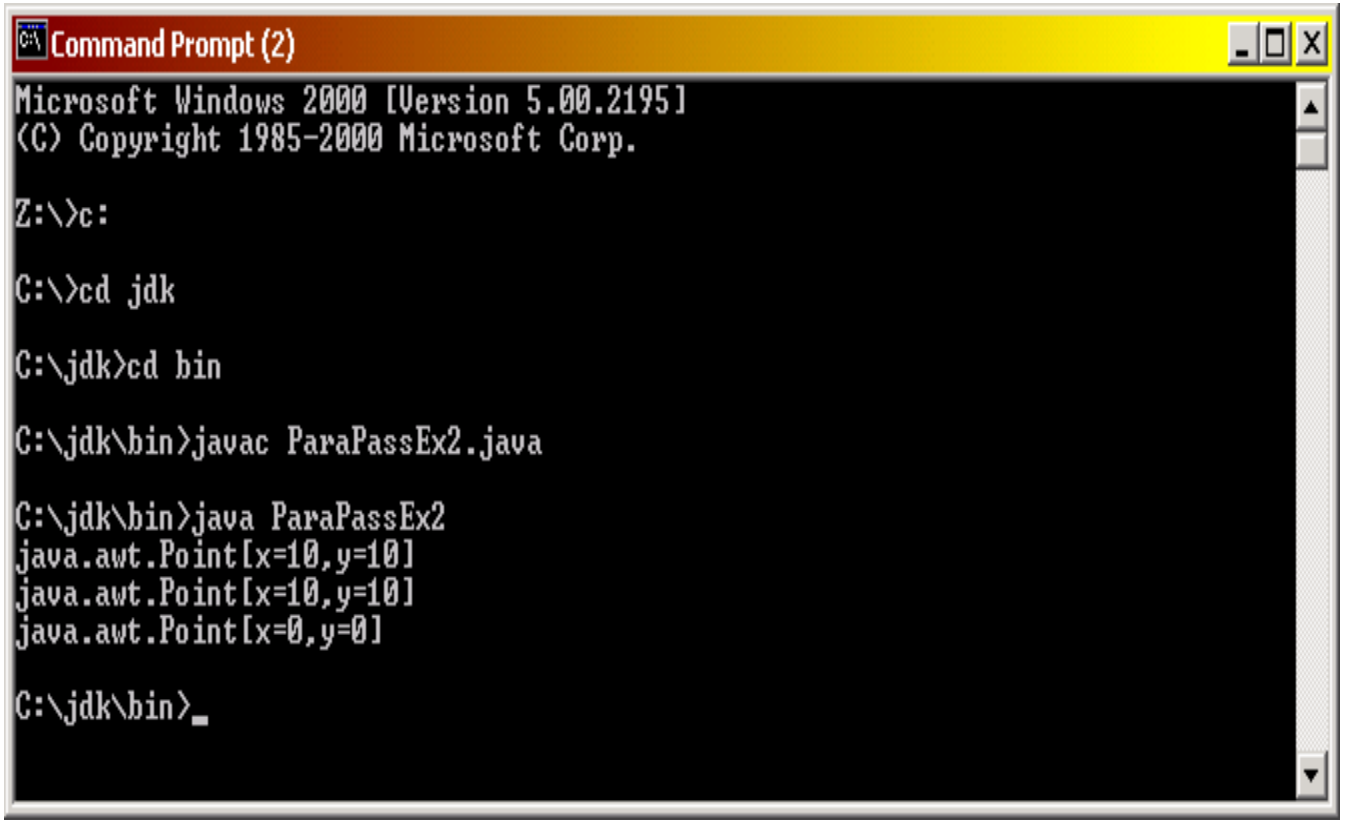
Within the multiply() method the return statement is evaluated. This evaluation sets the value of **x** to 88.0. The method's completion causes the memory associated with its activation record to be released. The flow of control is returned to the main() method, where the multiply() return value is used to initialize **`product`**. The main() method activation record now looks like the following:



Another Parameter Passing Example (2)

```
//Parameter passing example Number 2 for Day 9
import java.awt.*;
public class ParaPassEx2 {
   //f() set the value of the formal parameter
   public static void f(Point v){
        v = new Point(0,0);
   }
   //g() modify the contents of a referred object
   public static void g(Point v){
        v.SetLocation(0,0);
   }
   //main() application entry point
   public static void main(String[] args){
        Point p = new Point(10,10);
        System.out.println(p);
        f(p);
        System.out.println(p);
        g(p);
        System.out.println(p);
   }
}
```

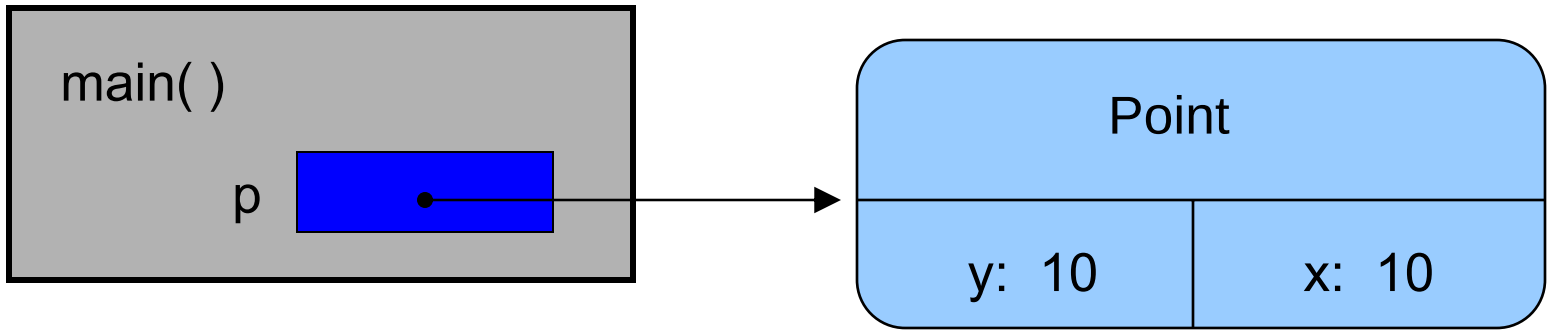

Another Parameter Passing Example – Output (2)



Another Parameter Passing Example (cont.)

Step 1:

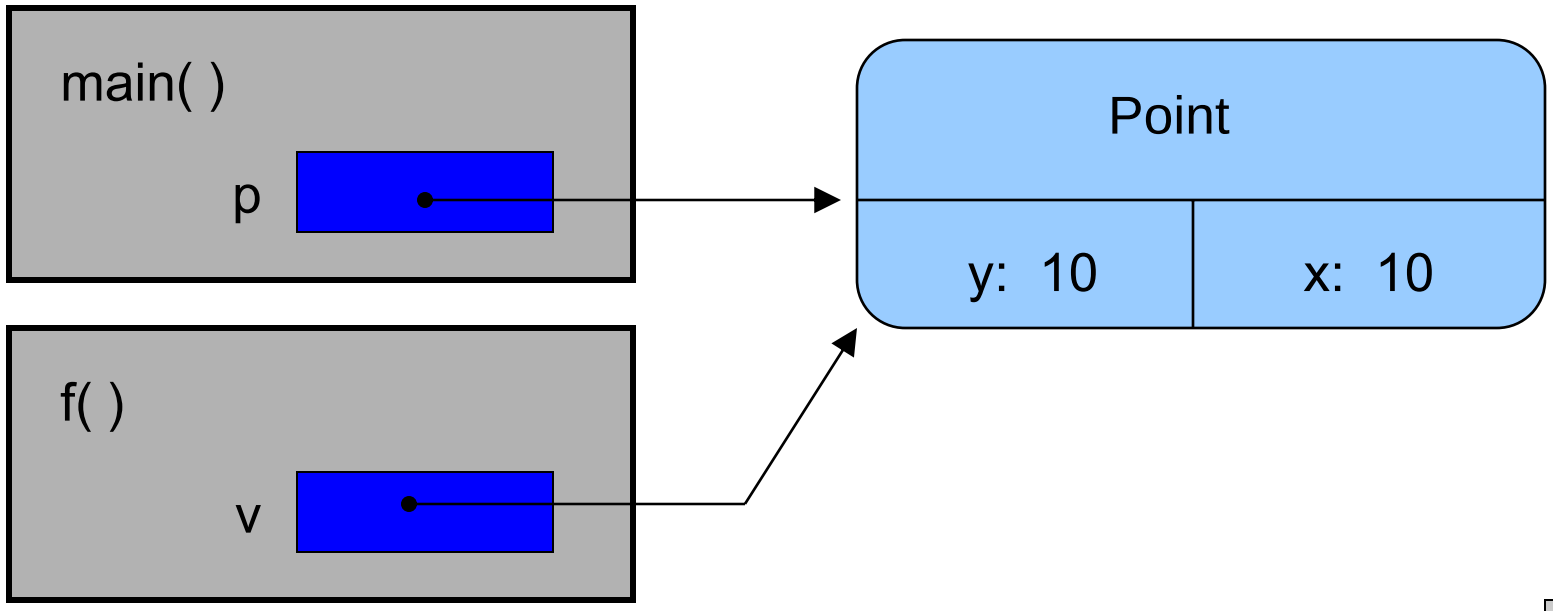
Main() method begins by initializing, and displaying a **`Point`** variable **`p`**. Variable **`p`** references an object referencing the location (10, 10). After completing this initial segment of main(), the activation record for main() is shown below. Although variable **`p`** is part of the activation record for main(), the memory for the Point to which it refers is not part of the activation record. Java divides the memory allocated to a program into two parts – the stack and the heap. Activation records are maintained in the stack and space for objects comes from the heap.



Another Parameter Passing Example (cont.)

Step 2:

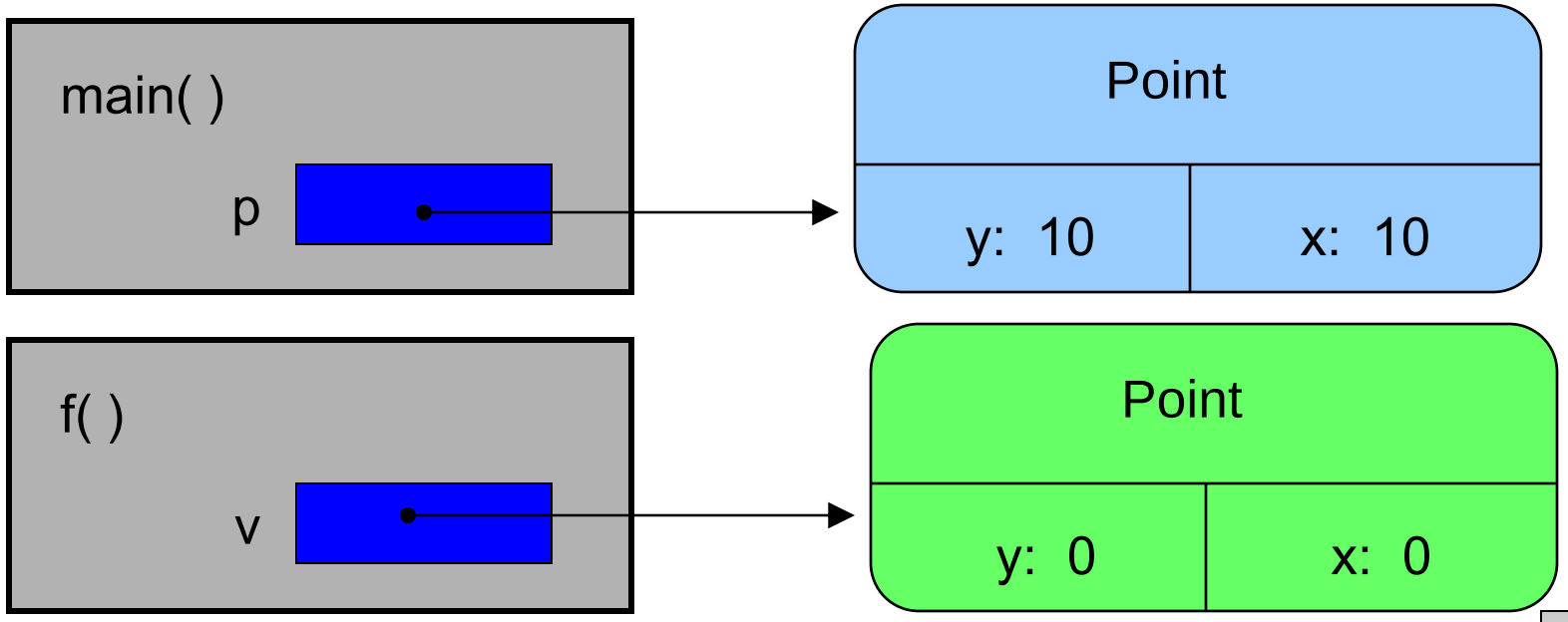
Method f() is then invoked with **p** as the actual parameter. The invocation causes the creation of an activation record and the flow of control is passed to the method. Main method variable **p** and formal parameter **v** of f() have the same value, which is a reference to an object representing location (10, 10). The activation records now looks like the following:



Another Parameter Passing Example (cont.)

Step 3:

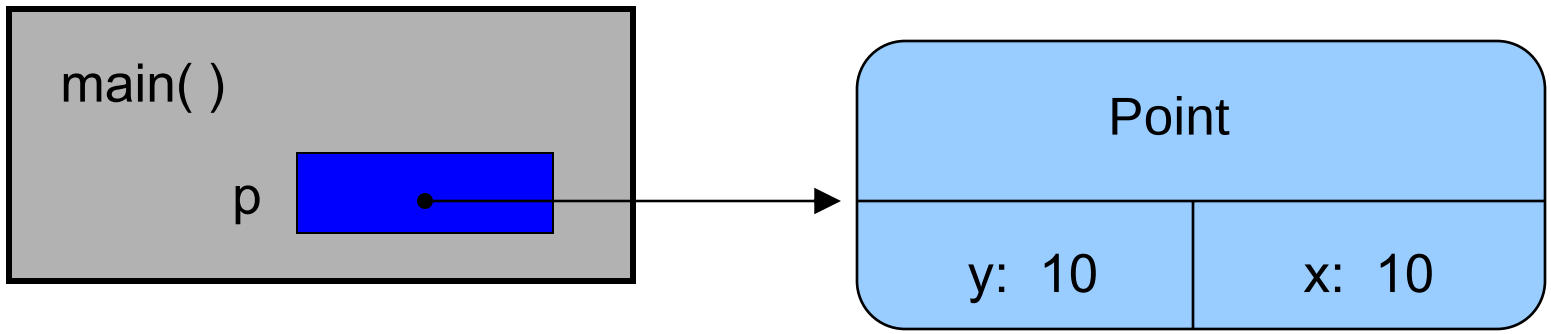
Since Java uses value parameter passing, the actual and formal parameters **p** and **v** have the same value, that is, they reference the same Point object. The assignment statement in method f() is then executed giving formal parameter **v** a new value, that is, **v** now references a new Point object representing the location (0, 0). This assignment does not modify main() method variable **p** because variable **p** has its own memory where it stores its value.



Another Parameter Passing Example (cont.)

Step 4:

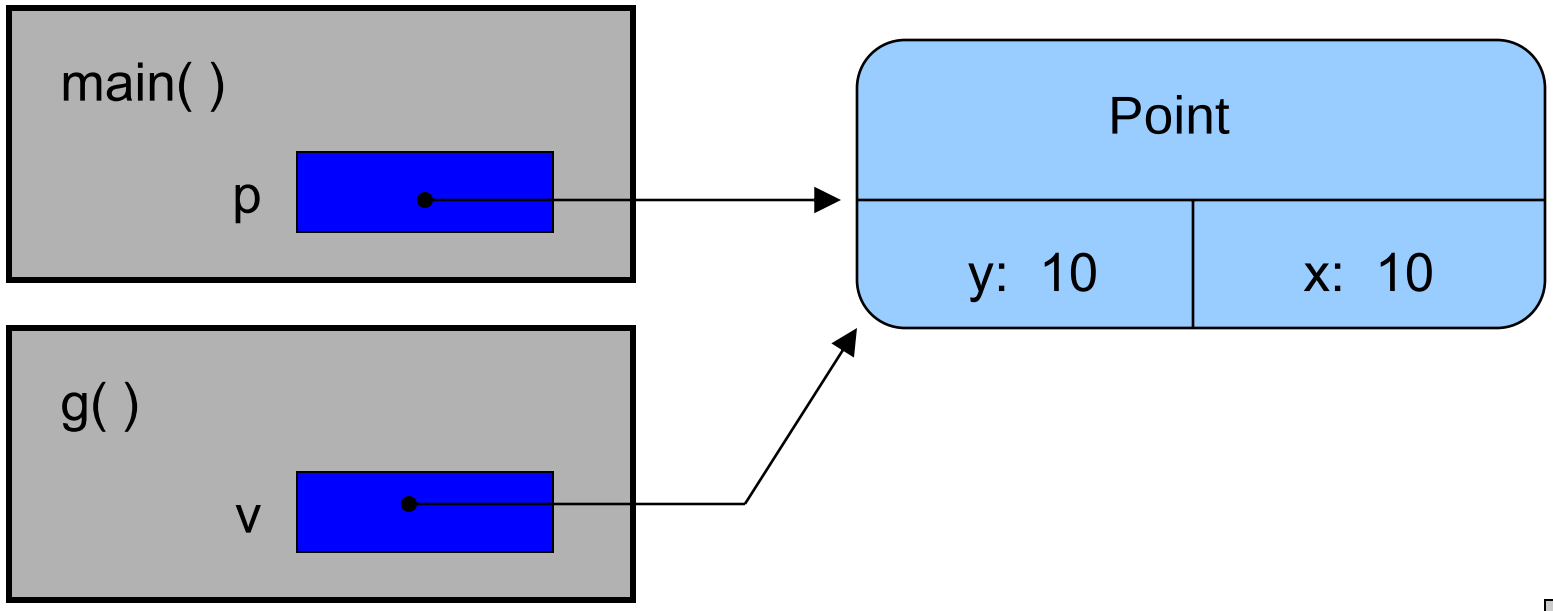
The activation record for f() is then released and flow of control is transferred back to the main() method where a println statement is executed. The activation record for main() looks like the following:



Another Parameter Passing Example (cont.)

Step 5:

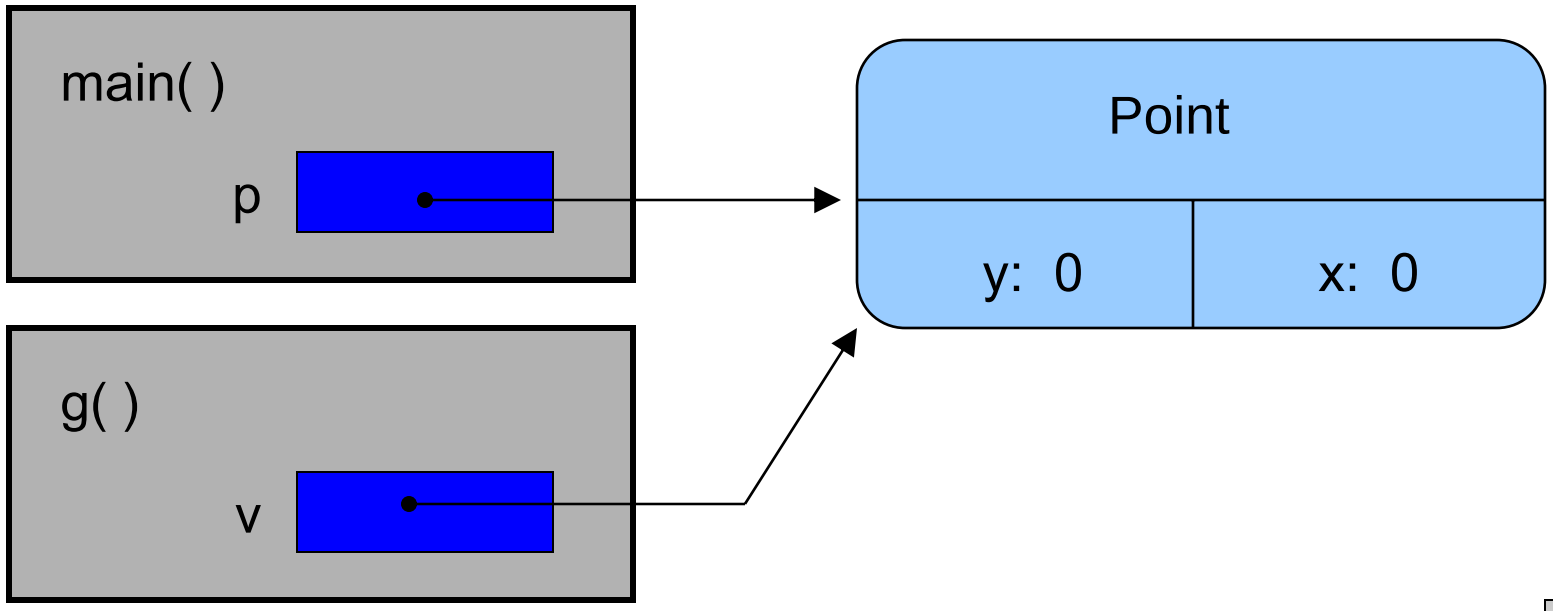
Method g() is then invoked with **p** as the actual parameter. The invocation causes the creation of an activation record and the flow of control is passed to the method. Main method variable **p** and formal parameter **v** of f() have the same value, which is a reference to an object representing location (10, 10). The activation records now looks like the following:



Another Parameter Passing Example (cont.)

Step 6:

Within method g() the Point instance method is invoked (v.setLocation(0,0)). This invocation causes an update to the Point object to which parameter v references. The update changes the representation to that of (0, 0). Notice that this update changed neither actual parameter p nor formal parameter v. However, the object to which they both refer has been updated and its change is visible through either one of them. The activation records now look like those shown below:



Summary of Parameter Passing in Java

- Java uses the ***pass by value*** method for parameter passing.
- The value of an actual parameter does not change with method invocation.
- If the actual parameter is a reference, then the object to which the actual parameter refers ***can*** be modified in a method invocation.

Object Reference `this`

- The keyword `this` can be used inside instance methods to refer to the *receiving* object of the method.
- The receiving object is the object through which the method is invoked.
- The object reference `this` cannot occur inside static methods.
- Two common usage of this:
  - to pass the receiving object as a parameter
  - to access fields shadowed by local variables.

- Each instance method runs under an object, and this object is accessible using the keyword `this`.

Passing `this` as a Parameter

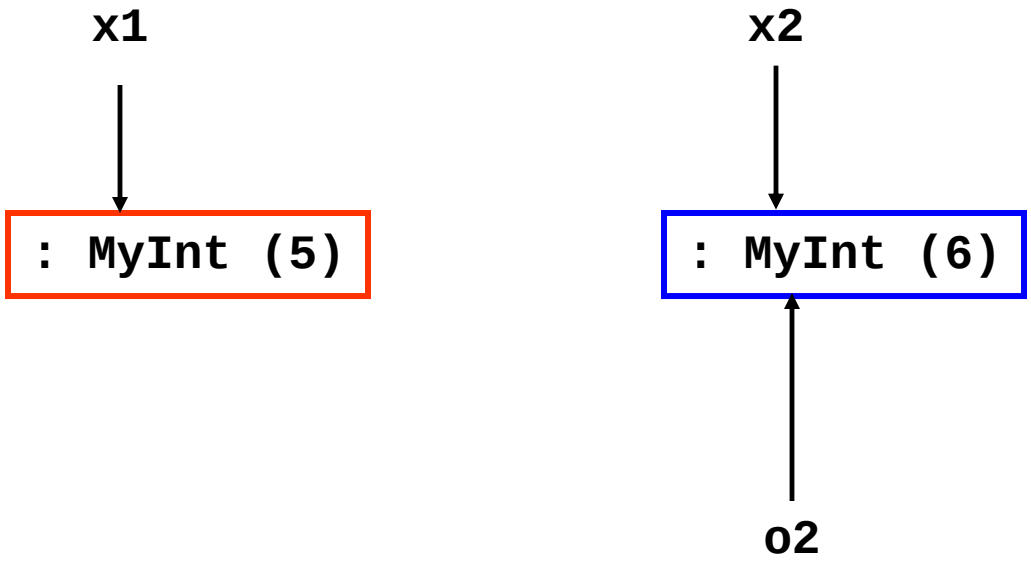
```
public class MyInt {
   private int ival;
   public MyInt(int val) { ival=val; }
   public boolean isGreaterThan(MyInt o2) {
       return (ival > o2.ival);
   }
   public boolean isLessThan(MyInt o2) {
       return (o2.isGreaterThan(this));
   }
}
```

Usage in some other place (method)

```
   MyInt x1=new MyInt(5), x2=new MyInt(6);
   x1.isGreaterThan(x2);  //output false
   x1.isLessThan(x2);     //output true
```


Passing `this` as a Parameter (cont.)

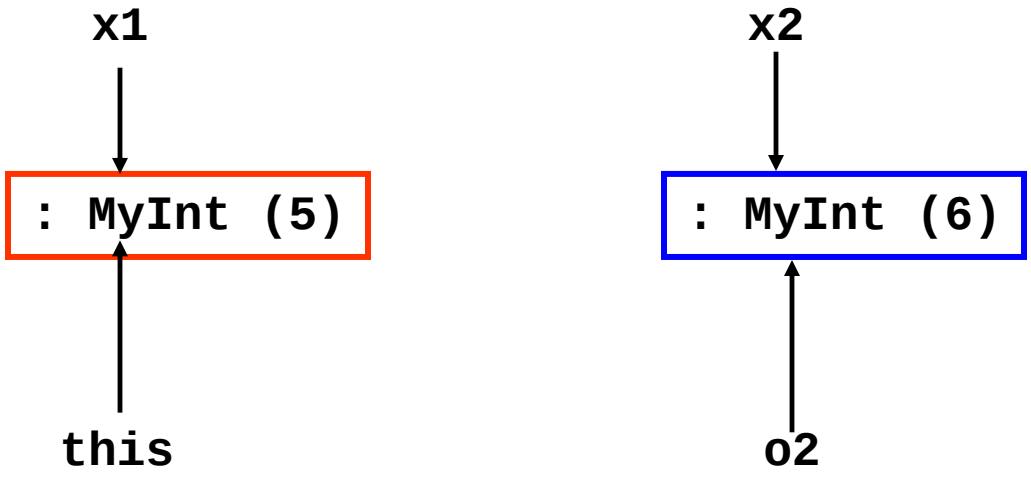
Variable bindings just after entering isGreaterThan method of x1



x1.ival is compared to x2.ival, since 5 is not greater than 6, false is returned

Passing `this` as a Parameter (cont.)

Variable bindings just after entering isLessThan method of x1



x1.isLessThan(x2) sets o2 = x2 and then calls o2.isGreaterThan(this)

`this` refers to the object on which the method invocation occurred which is x1. So the invocation is: o2.isGreaterThan(x1). On this invocation the formal parameter o2 becomes x1and what is returned is the result of the comparison of x2.ival > x1.ival (6 > 5) which is true.

Accessing Shadowed Fields

- A field declared in a class can be **shadowed** (hidden) in a method by a parameter or a local variable of the same name.

```
public class T {
   int x;   // an instance variable
   void m1(int x) { ... }   // x is shadowed by a parameter
   void m2() { int x; ... } // x is shadowed by a local variable
```

- To access a shadowed instance variable, we may use `this` keyword.

```
public class T {
        int x;   // an instance variable
        void changeX(int x) { this.x = x; }
}
```


A Closer Look At `this` As Well As Some of That

- The following few pages present another big Java example that will help illustrate the use of the keyword `this` along with several other features of Java that we have already seen as well as a few that we will be seeing in more detail in the near future.
- These new features that are included in the following example include inherited methods and overriding methods.
- We'll also introduce the concepts of facilitators, mutators, and accessors.

```
//Developer:  Mark Llewellyn          Date: June 2007
//Illustrate this and other Java features – creates a 3-tuple
public class Triple{
   //instance variables for the three attributes of the 3-tuple
   private int x1;     //first value
   private int x2;     //second value
   private int x3;     //third value

   //Triple() – default constructor
   public Triple(){
        this(0, 0, 0);
   ]

   //Triple() – specific constructor
   public Triple(int a, int b, int c) {
        setValue(1, a);
        setValue(2, b);
        setValue(3, c);
   }
```


```
    //getValue() – attribute accessor method
    public int getValue(int i) {
         switch (i) {
                  case 1: return x1;
                  case 2: return x2;
                  case 3: return x3;
                  default:
                           System.err.println("Triple: bad get: " + i);
                           System.exit(i);
                           return(i);
         }//end switch
    }

    //setValue() – attribute mutator
    public void setValue(int i, int value) {
         switch (i) {
                  case 1: x1 = value; return;
                  case 2: x2 = value; return;
                  case 3: x3 = value; return;
                  default:
                           System.err.println("Triple: bad set: " + i);
                           System.exit(i);
                           return(i);
         }//end switch
}
```


```
//toString() – string representation facilitator
public String toString() {
     int a = getValue(1);
     int b = getValue(2);
     int c = getValue(3);
     return “Triple[“ + a + “, “ + b + “, “ + c + “]”;
}

//clone() – duplicate facilitator
public Object clone() {
     int a = getValue(1);
     int b = getValue(2);
     int c = getValue(3);
     return new Triple(a, b, c);
}
```


```
    //equals() – equal facilitator
    public boolean equals(Object v) {
         if (v instanceof Triple) {
                  int a1 = getValue(1);
                  int b1 = getValue(2);
                  int c1 = getValue(3);

                  Triple t = (Triple) v;
                  int a2 = t.getValue(1);
                  int b2 = t.getValue(2);
                  int c2 = t.getValue(3);

                  return (a1 == a2) && (b1 == b2) && (c1 == c2);
         }
         else {
                  return false;
         }
    }
}//end class
```


Explanations For `Triple` Class

- Notice that there are two constructors for the `Triple` class. One takes 0 parameters and the other takes 3 integer parameters.
- The constructor requiring 0 parameters is called the ***default constructor***. The default constructor uses the specific constructor (the one requiring 3 parameters) to initialize a new `Triple` object.

```
public Triple(){
       this(0, 0, 0);}
```

The new `Triple` object (the `this` object) is constructed by invoking the `Triple` constructor expecting three integer values as actual parameters.

Explanations For `Triple` Class (cont.)

- Keyword `this` is a reference to the object being acted upon. It provides a notation for an object to refer to itself.
- In a constructor, `this` references the object under construction. (This is the case in our example.)
- In an instance method, `this` references the object being manipulated.
- The use of `this` in the `Triple` default constructor signifies that the object under construction is to be configured with zeros for its attribute values. The configuration is performed by invoking the `Triple` constructor whose signature matches the invocation, i.e., the `Triple` specific constructor expecting three integer values as parameters.

Explanations For `Triple` Class (cont.)

- This is a very common type of construction used to define constructors in Java. Typically, one constructor has a detailed initialization process. The other constructors then use that definition by invoking it with the appropriate values.
- By using that single detailed initialization process, it is less likely that some aspect of object configuration will be omitted.
- **NOTE**: If another constructor of the class assists the initialization, Java requires that the **this()** invocation be at the beginning of the statement body. The following constructor definition would be illegal:

```
public Triple() {
        int a = 0;
        int b = 0;
        int c = 0;
        this(a, b, c);
}
```


Explanations For `Triple` Class (cont.)

- The `Triple` specific constructor uses its three actual parameters to initialize the attributes of the `Triple` object under construction. It does so through the use of the `Triple` **mutators**.

```
//Triple() – specific constructor
        public Triple(int a, int b, int c) {
                setValue(1, a);
                setValue(2, b);
                setValue(3, c);
}
```


Explanations For `Triple` Class (cont.)

- The `Triple` specific constructor could have used explicit this references to indicate that it is the object under construction whose `setValues()` mutators are being invoked.

```
//Triple() – specific constructor
        public Triple(int a, int b, int c) {
                this.setValue(1, a);
                this.setValue(2, b);
                this.setValue(3, c);
}
```

- However, in practice Java programmers will tend generally to omit explicit `this` references. It is understood implicitly that the methods being invoked are acting upon the object under consideration.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- Every Java class (like the class `Triple` in this example) is automatically an ***extension*** of the standard class `Object`.
- The `Object` class specifies some basic behaviors common to all objects.
- One of the methods that `Triple` inherits from `Object` is the method `toString()`.
- **It is generally recommended that every class override the `toString()` definition provided by `Object` (i.e., provide a different implementation.)**
  - By doing so, `System.out.println()` can display a meaningful representation of all of the attributes of the object rather than just its `Object` attributes.
  - `System.out.println(p);` //display a string version of object p.
  - This can be most helpful when debugging a program.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- For the `Triple` class in this example, a reasonable string representation would be a listing of its three attributes.

```
//toString() – string representation facilitator
public String toString() {
        int a = getValue(1);
        int b = getValue(2);
        int c = getValue(3);
        return “Triple[“ + a + “, “ + b + “, “ + c “]”;
}
```


Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- Using the method from the previous slide, the code segment

```
Triple t1 = new Triple(10, 70, 30);
System.out.println(t1);
Triple t2 = new Triple(40, 50, 55);
System.out.println(t2);
```

Produces the following output:

```
Triple[10, 70, 30]
Triple[40, 50, 55]
```

- Notice that the name of the class is displayed as part of the representation. This is a common practice in overridden methods to provide debugging information.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- If a class does not override `toString()`, then the inherited method is used to produce its string representations.
- `Object` method `toString()` returns a string consisting of the class name appended with a character '@' and the hash code of the object. Such information is not particularly useful in most programming contexts.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- There are two other methods in class `Triple` that are overrides of methods inherited from the class `Object`. They are **`clone()`** and **`equals()`.**
- Method `clone()` returns a new `Triple` object that is a duplicate of the current object.

```
//clone() – duplicate facilitator
public Object clone() {
        int a = getValue(1);
        int b = getValue(2);
        int c = getValue(3);
        return new Triple(a, b, c);
}
```

- Because it is an override, the method has the return type of `Object`, which is the return type of the `clone()` method defined by the class `Object`.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- Creating a `Triple` clone is straightforward.
  - First variables `a, b`, and `c` are initialized with the attribute values of the `Triple` object that invoked method `clone()`.
  - The return value is produced by creating a new `Triple` object with attribute values `a, b`, and `c`.
- The following code segment will use the `clone()` method to initialize variable `t2`.

```
Triple t1 = new Triple(12, 14, 16);
Triple t2 = (Triple) t1.clone();
System.out.println("t1 = " + t1);
System.out.println("t2 = " + t2);
```

- Although the `Triple` method `clone()` returns a `Triple` object, its return type is `Object`. Therefore, the return value must be cast to `Triple` to use that value appropriately.

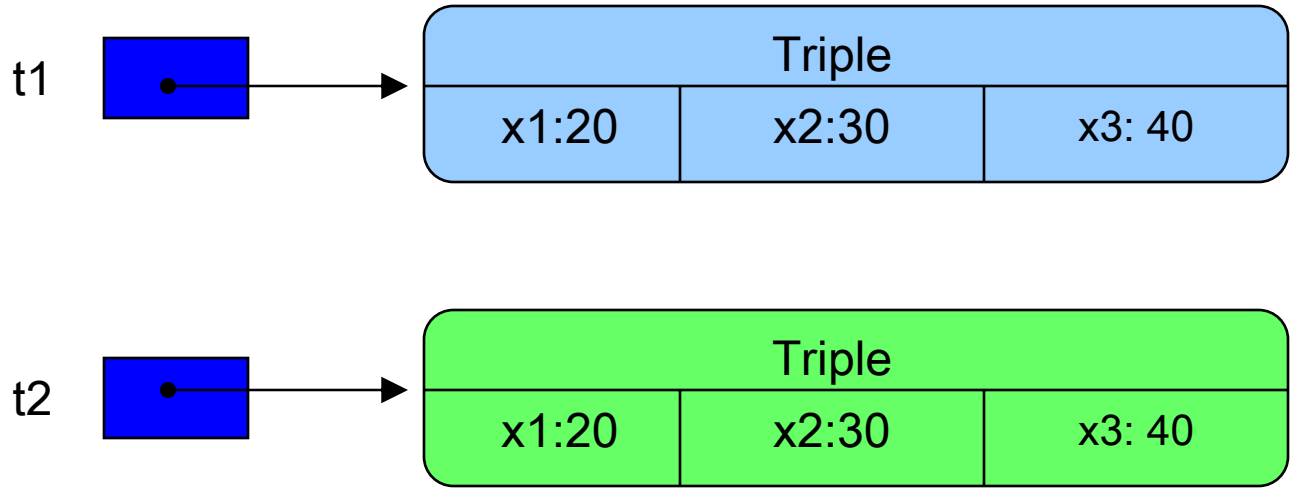
Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- The `Triple` class definition also overrides the `Object` class `equals()` method because the inherited `equals()` method does not meet our purposes here.
- Consider the following code segment that defines `Triple` variables `t1` and `t2` that reference similarly constructed objects:

```
Triple t1 = new Triple(20, 30, 40);
Triple t2 = new Triple(20, 30, 40);
```

- The representations of t1 and t2 are shown on the next slide.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)



- The inherited Object class equals() method reports if the two object references are the same rather than reporting whether the referenced objects have equivalent attributes.
- Thus, `System.out.println(t1.equals(t2));` will return **false** – variables t1 and t2 contain different references.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- `Object` method `equals()` is equivalent to the `==` operator. Both test whether the objects in question are in fact the same object.
- The `Triple` implementation of `equals()` instead determines whether its parameter is a `Triple` object with attributes that are equivalent to the invoking `Triple` object.
- The `Triple` implementation of the `equals()` method follows the standard form of an `equals()` method.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

```
//equals() – equal facilitator
   public boolean equals(Object v) {
        if (v instanceof Triple) {
                int a1 = getValue(1);
                int b1 = getValue(2);
                int c1 = getValue(3);
                Triple t = (Triple) v;
                int a2 = t.getValue(1);
                int b2 = t.getValue(2);
                int c2 = t.getValue(3);
                return (a1 == a2) && (b1 == b2) && (c1 == c2);
        }
        else {return false;}
   }
```

- The implementation begins by testing whether its parameter `v` references a `Triple` object. Operator **`instanceof`** returns true when the type of the object referenced by the left hand operand is either that of the right hand operand or derived from the right hand operand; otherwise the operator returns false. [NOTE: The **`null`** reference is never considered an instance of a class.]

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- If `v` is not referencing a `Triple` object, then the test expression evaluates to false and the else clause is executed – returning false as the object cannot be equal to the `Triple` object invoking the `equals()` method.
- If the test expression evaluates to true, then `v` is referencing a non-null `Triple` object. As such, the method can determine whether the corresponding attributes of the two `Triple` objects match.
- Variables `a1, b1, c1, a2, b2,` and `c2` represent respectively the attributes of the object invoking the `equals()` method and the `Triple` object referenced by parameter `v`.

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

```
int a1 = getValue(1);
int b1 = getValue(2);
int c1 = getValue(3);
Triple t = (Triple) v;
int a2 = t.getValue(1);
int b2 = t.getValue(2);
int c2 = t.getValue(3);
```

- ? Why does the code define a variable `t` that is a cast of `v`?
  - Although it must be the case at this point in the method that parameter `v` references a `Triple` object (we've already tested for this), Java still requires an explicit case to treat `v` as something other than an `Object`.
  - Therefore, the statement **`int a2 = v.getValue(1);`**

    will not compile because the apparent type of the object referenced by `v` is `Object`, which does not have a method `getValue()`.

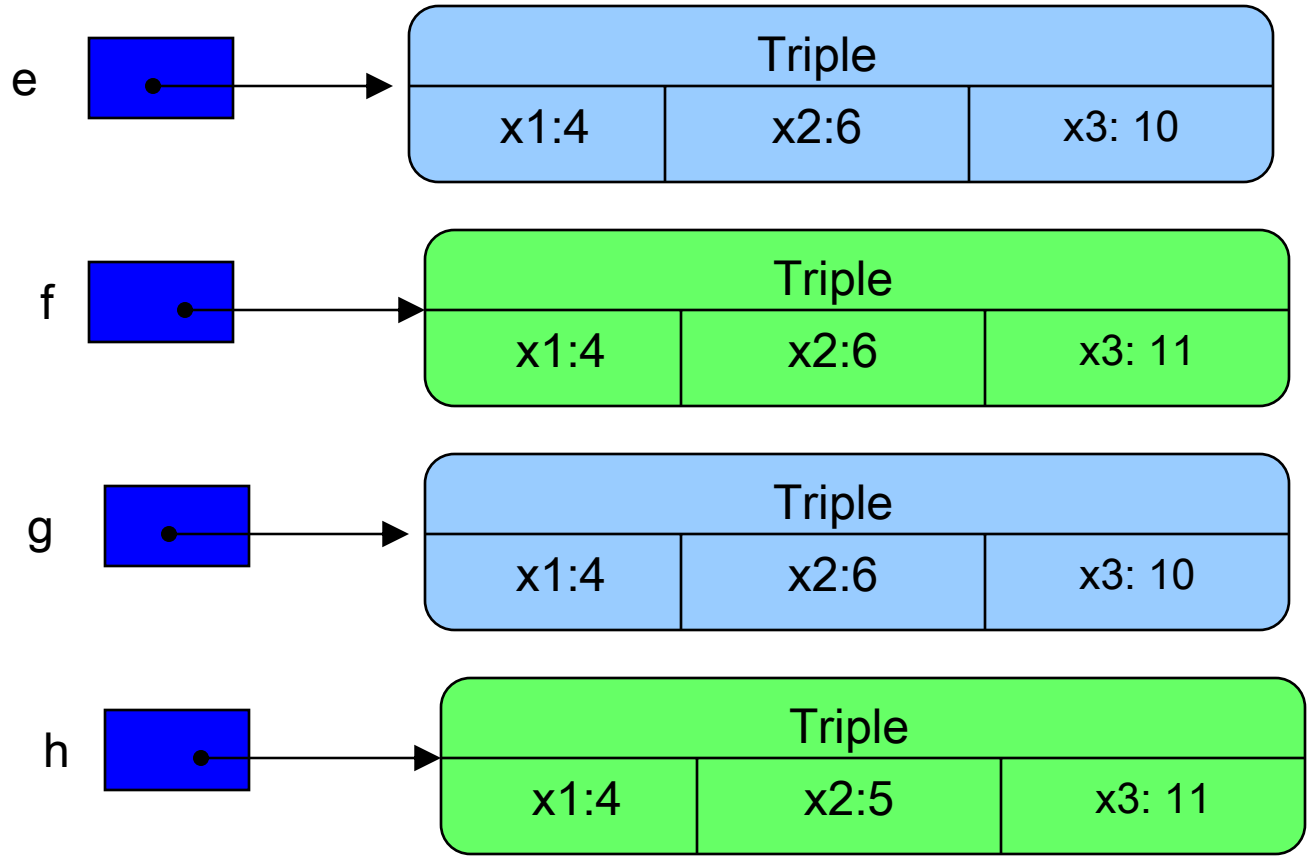
Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

- Consider the following code segment:

```
Triple e = new Triple(4, 6, 10);
Triple f = new Triple(4, 6, 11);
Triple g = new Triple(4, 6, 10);
Triple h = new Triple(4, 5, 11);
boolean flag1 = e.equals(f);
boolean flag2 = e.equals(g);
boolean flag3 = g.equals(h);
```

- What are the values assigned to `flag1, flag2`, and `flag3` by this code? Try it before you go to the next slide!

Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)



Inherited Methods and Overriding - Explanations For `Triple` Class (cont.)

The objects referred to by e and f are different because their x3 attributes are different (10 ≠ 11). The objects referred to by e and g are equivalent because the corresponding attribute values are the same. The objects referred to by g and h are different because their x2 attributes are different (6 ≠ 5).

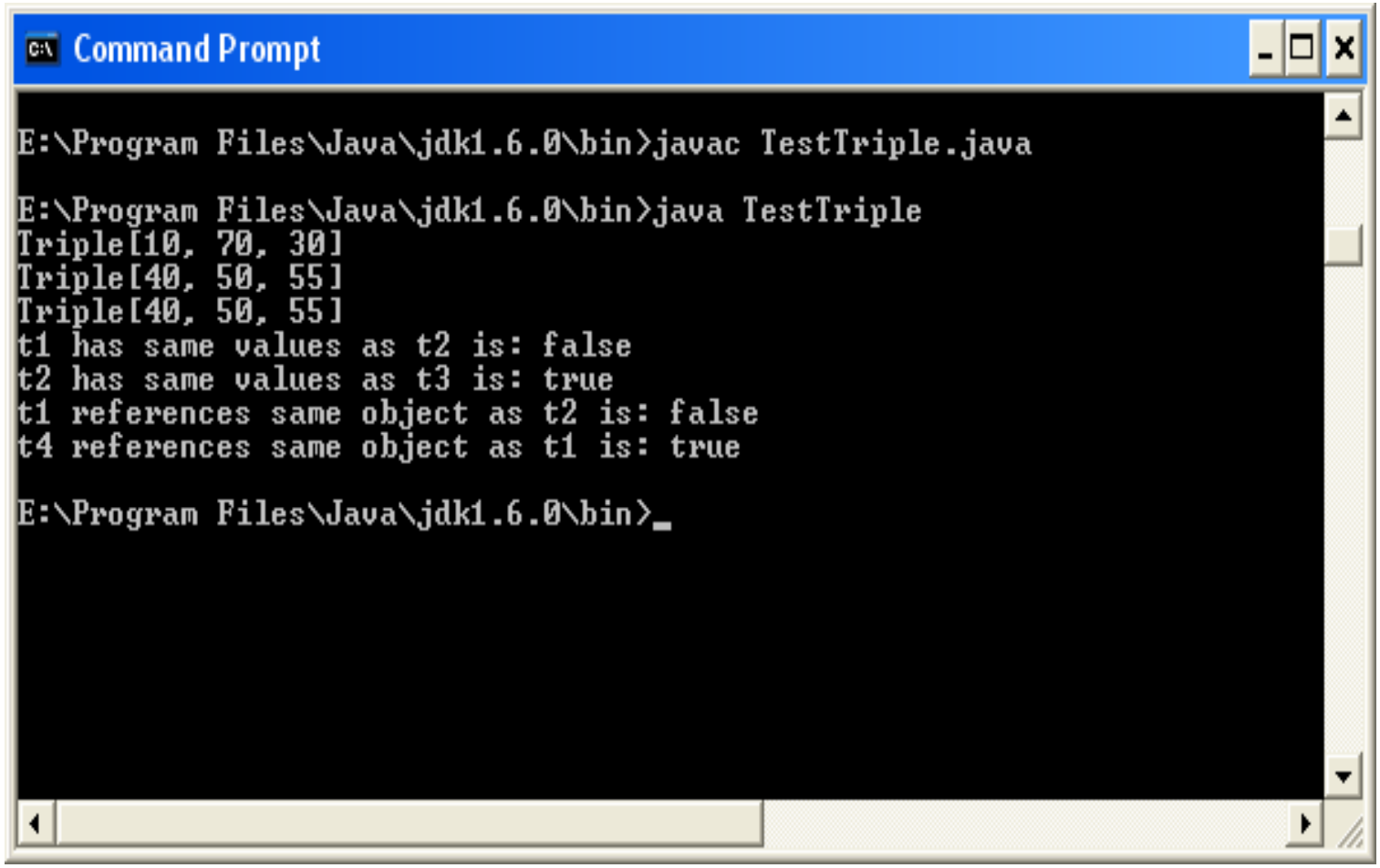
Untitled - Notepad

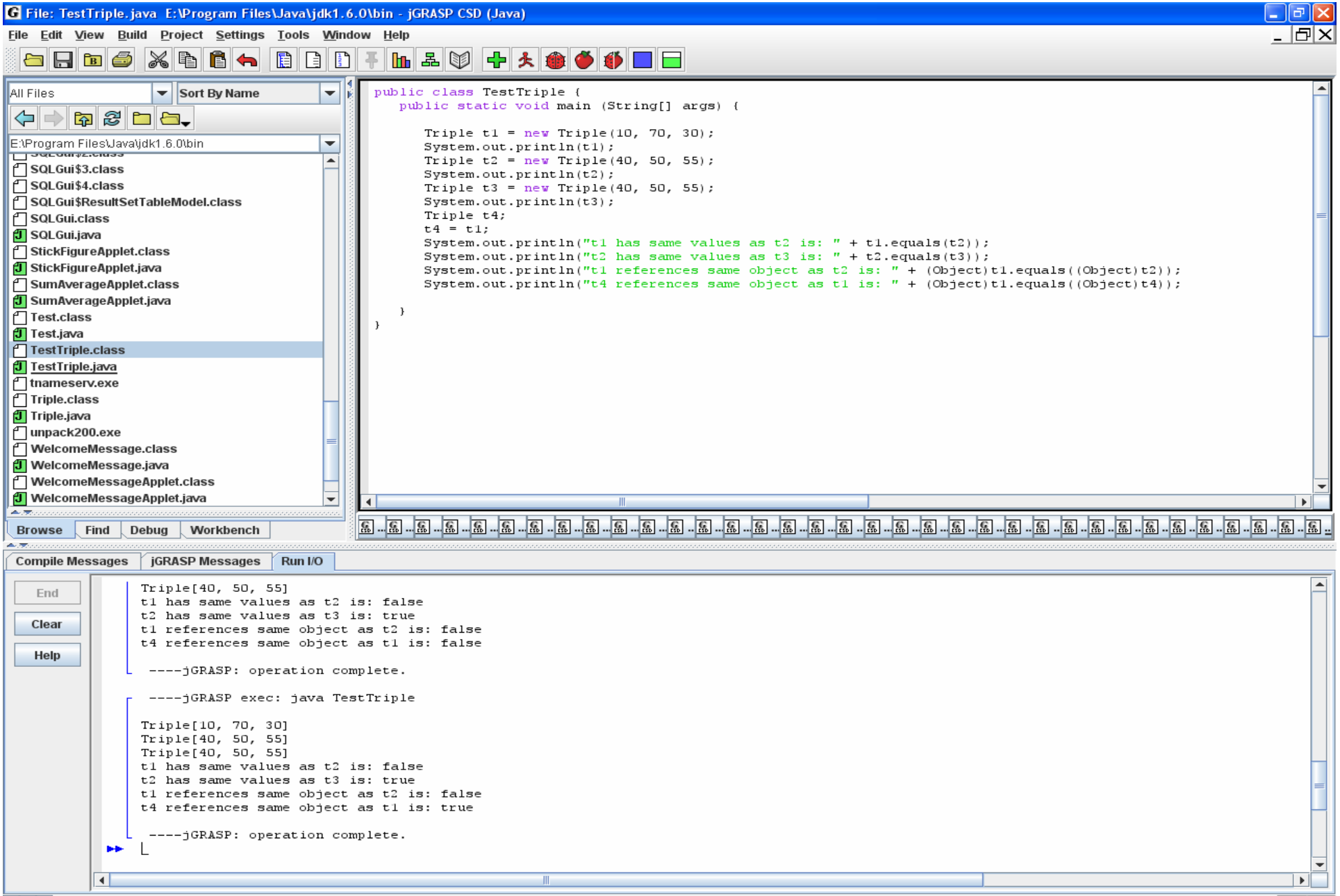
File Edit Format View Help

```
public class TestTriple {
        public static void main (String[] args) {

                Triple t1 = new Triple(10, 70, 30);
                System.out.println(t1);
                Triple t2 = new Triple(40, 50, 55);
                System.out.println(t2);
                Triple t3 = new Triple(40, 50, 55);
                System.out.println(t3);
                Triple t4;
                t4 = t1;
                System.out.println("t1 has same values as t2 is: " + t1.equals(t2));
                System.out.println("t2 has same values as t3 is: " + t2.equals(t3));
                System.out.println("t1 references same object as t2 is: " + (Object)t1.equals((Object)t2));
                System.out.println("t4 references same object as t1 is: " + (Object)t1.equals((Object)t4));

        }
}
```



Command Prompt
E:\Program Files\Java\jdk1.6.0\bin>javac TestTriple.java
E:\Program Files\Java\jdk1.6.0\bin>java TestTriple
Triple[10, 70, 30]
Triple[40, 50, 55]
Triple[40, 50, 55]
t1 has same values as t2 is: false
t2 has same values as t3 is: true
t1 references same object as t2 is: false
t4 references same object as t1 is: true
E:\Program Files\Java\jdk1.6.0\bin>


File: TestTriple.java E:\Program Files\Java\jdk1.6.0\bin - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
All Files
Sort By Name
E:\Program Files\Java\jdk1.6.0\bin
SQLGui$3.class
SQLGui$4.class
SQLGui$ResultSetTableModel.class
SQLGui.class
SQLGui.java
StickFigureApplet.class
StickFigureApplet.java
SumAverageApplet.class
SumAverageApplet.java
Test.class
Test.java
TestTriple.class
TestTriple.java
tnameserv.exe
Triple.class
Triple.java
unpack200.exe
WelcomeMessage.class
WelcomeMessage.java
WelcomeMessageApplet.class
WelcomeMessageApplet.java
Browse Find Debug Workbench
public class TestTriple {
public static void main (String[] args) {
Triple t1 = new Triple(10, 70, 30);
System.out.println(t1);
Triple t2 = new Triple(40, 50, 55);
System.out.println(t2);
Triple t3 = new Triple(40, 50, 55);
System.out.println(t3);
Triple t4;
t4 = t1;
System.out.println("t1 has same values as t2 is: " + t1.equals(t2));
System.out.println("t2 has same values as t3 is: " + t2.equals(t3));
System.out.println("t1 references same object as t2 is: " + (Object)t1.equals((Object)t2));
System.out.println("t4 references same object as t1 is: " + (Object)t1.equals((Object)t4));
}
}
Compile Messages jGRASP Messages Run I/O
End
Clear
Help
Triple[40, 50, 55]
t1 has same values as t2 is: false
t2 has same values as t3 is: true
t1 references same object as t2 is: false
t4 references same object as t1 is: false
----jGRASP: operation complete.
----jGRASP exec: java TestTriple
Triple[10, 70, 30]
Triple[40, 50, 55]
Triple[40, 50, 55]
t1 has same values as t2 is: false
t2 has same values as t3 is: true
t1 references same object as t2 is: false
t4 references same object as t1 is: true
----jGRASP: operation complete.
Line:18 Col:2 Code:0 Top:1